IDENTIFYING CRITICAL LOADS AND A STUDY OF THEIR ACCESS PATTERNS

CS752 PROJECT REPORT

ANURAG GUPTA, VISHAL KATHURIA

{ anurag , vishal }@cs.wisc.edu

Computer Sciences Department
University of Wisconsin, Madison
Madison – 53706, WI

December 15, 1998

ABSTRACT

The contribution of this report is an analysis of the access patterns of the loads that are critical to the performance of an out of order processor (henceforth called the Critical Loads). Our measurements reveal that 30%-40% (an in some cases, upto 98%) of the critical load instructions access the same memory location they accessed the last time they were executed. Moreover, in more than 80% of the cases, the successive occurrences of a critical load instruction in the dynamic instruction stream are atleast 512 instructions apart. On the basis of above analysis, we present a memory management technique to decrease the latency of critical loads and speedup the overall performance of an out of order processor.

1. INTRODUCTION

There has been a tremendous improvement in the processor performance over the last couple of decades. Today, most processors are capable of extremely high clock rates with complete out of order execution resulting in high performance. But on the other hand, improvement in the main memory performance has not kept pace with the processor performance. For example, processor performance has been increasing at almost 50% per year whereas memory access times have been improving at only 5-10% per year only **[2]**. As a result, there is a wide gap between the memory and processor performances. A balance should be maintained between the two, which unfortunately has not happened over the years. As a result, the memory access time due to a cache miss is likely to become a major bottleneck in the future.

Looking at the other face of the coin, as mentioned earlier, most modern processors are capable of out of order execution. They can buffer the instructions that are waiting for their operands from memory. So, if an instruction is waiting for its data from the main memory, then the processor can execute other independent instructions and still achieve high performance. *So, why bother to speed up the critical instructions?*

On close examination, we see that although the processor can use dynamic scheduling and buffering to execute other independent instructions, it is not always possible to find independent instructions to execute. This will result in the processor to stall, thereby reducing performance. Secondly, although the processor may employ sophisticated branch prediction mechanisms and allows speculative execution to execute instructions out of order, it must always commit the instructions in order. Thus, the finite resources of the processors may also cause it to stall. For example, the reorder buffer and the load/store queues have a finite size, if they are filled up waiting for an instruction at the head of the list, then the processor must stall. Thirdly, although most processors employ sophisticated branch prediction schemes, it mispredicts many times too. If a critical load is feeding a

mispredicted branch, then it will cause the processor to go down the wrong execution path. This will have an effect that although the processor is kept busy, no useful work is done.

Thus, it becomes important to identify such critical loads and provide mechanisms to speed up their execution so that they don't have to wait too long for their operands from memory, thereby allowing instructions that depend on them for their operands not to stall and improve processor performance.

We present a study of the access patterns of these critical loads (including a study of their spatial and temporal locality). We also present a hardware scheme that could be employed to decrease the latency of these critical loads. The study of the access patterns of these Critical Loads presents some interesting results, which we discuss and utilize to propose the hardware schemes.

The report is organized as follows. In **Section 2**, we will discuss what we mean by Critical Loads and how we can categorize loads as critical. In **Section 3**, we discuss the mechanism we have employed to identify these critical loads. **Section 4** discusses the implementation details. **Section 5** illustrates the results and observations. We interpret the results obtained and make some observations about the spatial and temporal locality of the critical loads. In **Section 6** we propose a hardware scheme based on the study of the access patterns of these critical loads. In **Section 7** we summarize our work done and point out some of the limitations of our study and in **Section 8**, we conclude with a mention of some of the future directions.

2. BACKGROUND

In the execution stream of instructions, there are certain instructions on which other instructions are dependent for their data values. The data values from memory for these instructions should be fetched as quickly as possible so that the instruction does not have to wait too long. These instructions are ***Critical*** and the processor has a ***Low Latency Tolerance*** for them.

We categorize loads as Critical which fall in the following two categories **[1]**:

1. *The loads on which the branches, especially the ones which are mispredicted depend, and*
2. *The loads on which a large number of succeeding instructions depend.*

Thus, all loads in an execution stream of a program can be categorized into the above two categories. If a branch depends on the operand from the load instruction then it must be computed as soon as possible. Also, if there are a large number of instructions that directly or indirectly depend on this load, then this load becomes

critical and needs to be completed quickly otherwise the processor might not be able to tolerate its latency and stall. Hence, these loads becomes critical to the processor performance and it becomes very important to identify these loads and service them as fast as possible. In the next section we describe the scheme that we use to identify the critical loads.

3. IDENTIFICATION OF THE CRITICAL LOADS

For identifying the critical loads, we employ the following scheme, which we call the *Critical Load Identification Scheme* (**CLIS**):

We maintain an instruction window of the last N executed instructions. Instructions enter the window from one end and leave at the other end. Each entry in the Instruction Window contains the instruction and a pointer to a list of pointers. Each node in the list of pointers points to the head of the Dependency List to which it belongs. The head of each Dependency List contains a load instruction. A load can be identified to be critical based on its dependency list.

We now describe the three important data structures that we maintain for the implementation of this scheme:

3.1 Dependency List

The Dependency List is a list of instructions that are dependent on a load such that the instruction appearing later in the list has a direct or indirect data dependence on the instruction appearing earlier in the list. Since we are interested in instructions dependent on loads, each Dependency List has a load instruction at its head.

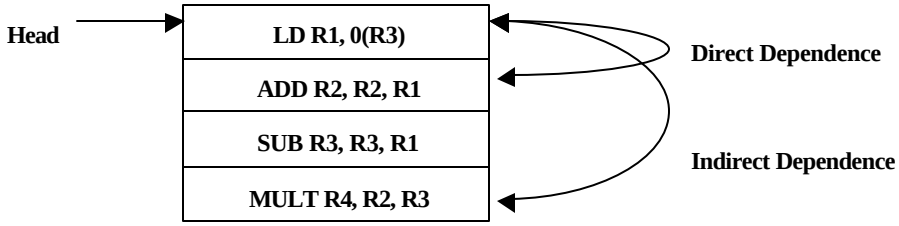

Fig. 1.0 Dependency List

Thus, as shown in **Fig. 1.0**, the Add instruction has a direct data dependence on the load instruction because one of its operand (*R1*) is dependent on the load instruction. Also, the Mult instruction has indirect data dependence through the Add instruction on the load instruction (*through operand R2*). Thus, the list contains all instructions that are directly or indirectly dependent on the load instruction. A Dependency List is maintained for all the load instructions.

Each entry in the dependency list contains the following information:

1. The dependent instruction
2. If the instruction was a branch, was it mispredicted
3. If the instruction was a load, did it hit/miss in the cache
4. The Program Counter (*PC*) of the instruction
5. The Dynamic Instruction Count (*DIC*) of the instruction

The Dynamic Instruction Count (DIC) of an instruction is the total number of instructions executed before that instruction. It is a unique value of any instruction.

Thus, a dependency list is created for each load instruction. An important point to note is that one particular instruction can be in more than one dependency list because it can depend on more than one load instruction for its source operand values.

3. 2 The Instruction Window

The Instruction Window contains the last N instructions that were executed before a current instruction. Each entry of the window corresponds to an instruction and also contains a pointer to a list of pointers to all the

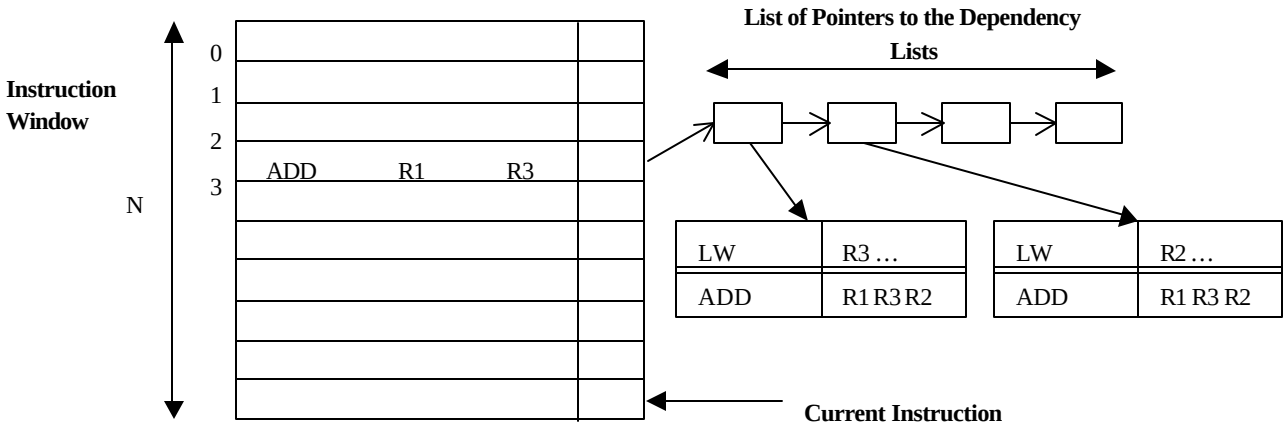

Fig. 2.0 Instruction Window

Dependency Lists to which the instruction belongs.

3.3 Register Table

We also maintain a Register Table, which has an entry corresponding to each of the architectural registers in the processor and a tag. This table is maintained to keep track of which last instruction last wrote to a register, thereby keeping track of the data dependencies of an instruction. Each entry of the table also contains a tag, which is an index into the instruction window. If the tag is a valid entry, then it corresponds to the

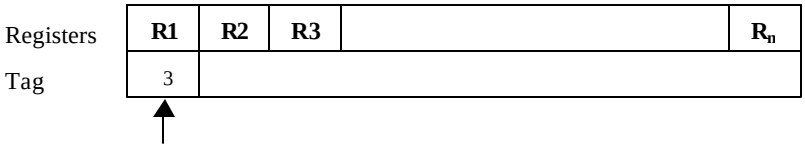

Fig. 3.0 Register Table

instruction that will write to that register.

For example, the tag in the **Fig. 3.0** contains the value 3, which is used to index the instruction window to find the instruction that will write to register R1. The corresponding entry in the instruction contains the Add instruction that will write to register R1 and also a pointer to a list of pointers. Each node in the list points to the dependency list to which this instruction belongs. This Add instruction belongs to two dependency lists because both its operand depend on different load instructions for its value.

Now, whenever a new instruction enters the instruction window, it is processed according to the following algorithm, presented in pseudo-code:

```
// Let I be the instruction currently being executed

for each (source operand s of I)
{
        tag = RT(s);        // RT is the Register Table
                            // Tag references to the instruction that last wrote to s

        if tag is invalid
                continue;           // that source operand does not depend on the load instruction

        // IW is the instruction window
        for each (Dependency List DL such that IW(tag) belongs to DL) {
                append I to DL;
        }
}
```

For example, if another instruction now enters the instruction window, which has a source operand as R1, first the tag will be used to index the instruction in the instruction window that will provide the value of R1. This, current instruction is thus dependent on the Add instruction, so it will be added to all the dependency lists of that instruction by going through the list of pointers to the dependency lists.

3.4 Maturation of Dependency List

A dependency list is said to be mature when any of the following conditions is satisfied:

1. The size of the dependency list exceeds a certain threshold value
2. The last instruction in the list has moved out of the instruction window. In this case, we know that no more instructions will be added to the list
3. The Dynamic Instruction Count of the current instruction minus the Dynamic Instruction Count of the load at the head of the list is greater than a certain threshold

The first case corresponds to the fact that *a lot of instructions* are dependent on the load at the head of the list and thus, the load must be critical. The second case states that if an instruction moves out of the instruction window, it moves out of focus and we can always say that no more instructions will be added to the list and thus the list can be matured. The third case corresponds to the impact range of the load instruction that is at the head of the dependency list since it is a count of the number of instructions executed between the two instructions in consideration.

Now, when a dependency list matures, it is sent to a function that determines if the load at the head of the list is critical or not. This function determines if the load at the head of the list is critical or not based on the following criteria:

1. If a mispredicted branch occurs in the dependency list, then the load is critical
2. If the size of the list is greater than a certain threshold, implying that a lot of instructions are dependent on that load, then it is identified to be a critical load

Thus, using the above scheme we identify all the Critical Loads in the dynamic execution stream of a program. The next section gives some details for the implementation of the above scheme.

4. IMPLEMENTATION

For the access pattern analysis, the following information is required.

1. The Critical Loads
2. Did the Load Hit in L1?
3. Address of the Data read by the Load.
4. Did the branch instruction dependent on this load get mispredicted?

For obtaining the above information, the functional simulators *sim-cache* and *sim-bpred* from the *SimpleScalar* tool set were modified to make a functional simulator *sim-cbp* that does the cache simulation as well as branch prediction. The above information is tapped from the simulator and is passed on to other functions, which maintain the Dependency Lists, Instruction Window and the mechanism to detect the Critical Loads. The Spec95 benchmark suite was used as inputs to the modified functional simulator. The *criticality* criterion is applied to every dynamic occurrence of a load and the following information about every load is output to a trace file.

1. Dynamic Instruction Count
2. PC (of the critical load)
3. Memory Address (of the data accessed by the load)
4. L1 / L2 miss

This trace file is then used to gather the following information for each PC (i.e. the PC of a critical load instruction)

1. Average Address Difference
2. Average Instruction Gap

4.1 Average Address Difference

The Address Difference for a load instruction is the difference in accessed addresses between the current execution of this PC and its last execution. This difference averaged through the whole execution is termed as Average Address Difference. This is the measure of spatial locality of the references made by the *same* load instruction.

4.2 Average Instruction Gap

The Instruction Gap is the number if instructions executed between the current execution of this PC and its last execution. This difference averaged through the whole execution is termed as Average Instruction Gap. This is a measure of temporal locality of the loads (i.e. PC's not the temporal locality of the data addresses).

The above two parameters gave the access patterns of the Critical Loads. The next section presents the results that we obtained and discusses t he observations that we made from them.

5. RESULTS AND OBSERVATIONS

The simulations were run for two L1 cache configurations viz. 8KB 1-way and 8 KB 2-way. In **Fig 5.1** and **Fig. 5.2** we present the graphs for the spatial locality for these caches. On the x-axis are the ranges of the Average Address Differences. On the y-axis is the percentage of Critical Load instructions (PC's) that had that Average Address Difference.

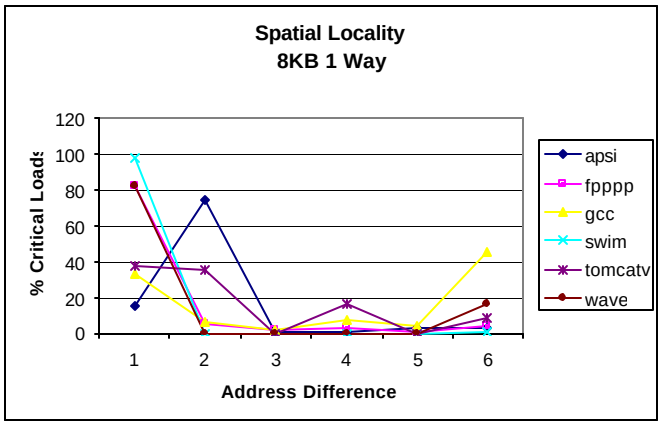

Fig. 5.1

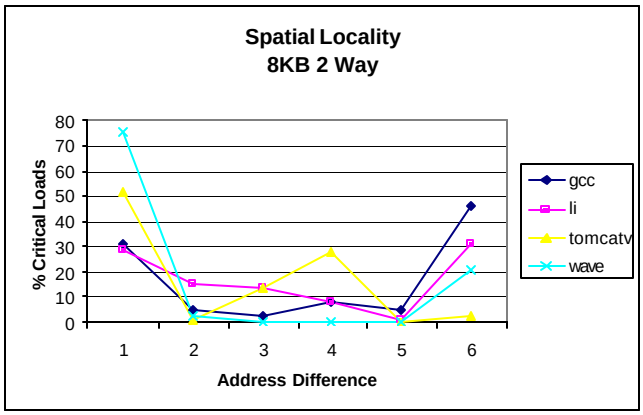

Fig. 5.2

The graphs show that a significant fraction of the Critical Loads in all the benchmarks are accessing the same memory location that they accessed earlier. This is particularly true of the floating-point benchmarks. We call these Critical Loads as Category 1 Critical Loads and the loads with an Address Difference > 512 are termed as Category 2 loads. The rest of the loads are being ignored for rest of the discussion because the loads that fall in these two categories form the majority of the Critical Loads. There is no significant difference in the percentage of Category 1 loads in the graphs for 1 way and 2 way caches. If the primary reason for these critical loads missing in L1 cache had been conflict misses, there should have been an appreciable decrease in the percentage of Category 1 Critical Loads. Since this is not the case, we conclude that any solution aimed at decreasing conflict misses (e.g. higher associativity, victim caches, etc) will not have a significant impact on the latency of critical loads.

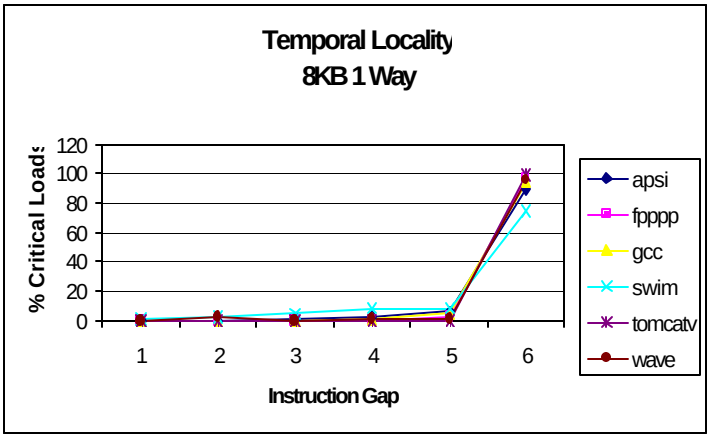

Fig. 5.3

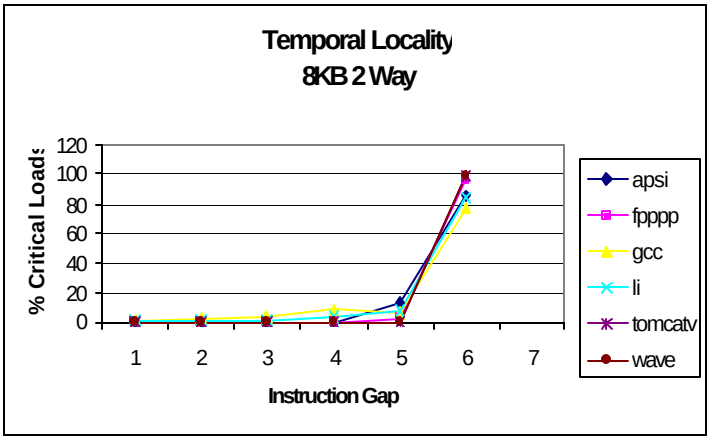

Fig. 5.4

Since all the loads used for this graph had missed in L1 cache, it can be expected that these category 1 critical loads are not occurring very frequently (otherwise their data would not have been thrown out of the L1 cache). This is corroborated by the graphs in **Fig. 5.3** and **Fig. 5.4** for the temporal locality of the *Category 1 Critical Loads*. On the x-axis is the Average Instruction Gap (AIG) and on the y-axis is the percentage of Category 1 Critical Loads with their AIG in that range. It is evident that most of the Category 1 Critical Loads (C1CL) appear again in the dynamic instruction stream at an interval > 512 instructions. Hence we conclude that *C1CL have low temporal locality.*

6. EXPLORING THE DESIGN SPACE

A high spatial locality exhibited by a significant fraction of the Critical Loads suggests that the memory management schemes designed to keep the data last accessed by Category 1 Critical Loads (*henceforth called Critical Data*), close to CPU, could help in decreasing C1CL latency. There are two alternate ways to achieve this:

1. Pin the critical data in the cache
2. Store the critical data in a separate buffer.

6.1 Pin the Critical Data in Cache

- ***Advantages***
  - Doesn't require too many extra transistors to implement.
- ***Disadvantages***
  - A policy of unpinning the pinned data is required. Aging is one alternative. Another alternative has been suggested by Alvin Lebeck **[5]**.
  - Since C1CL instructions do not repeat very often, the critical data might still get thrown away before it is still needed by a critical load.
  - The critical data eats up L1 cache space and can lead to performance loss because of non-critical data missing in L1 cache.

6.2 Store the critical data in a separate buffer

- **Advantages**
  - This scheme does not adversely affect the non-critical data hit rate
  - No overhead of maintaining a complex aging scheme
- **Disadvantage**
  - Will require more real estate on chip.

Some of the issues involved in this scheme are:

6.2.1 Buffer Thrashing (Size of the buffer? Associativity?)

The size of the buffer must be comparable to the size of the critical data set accessed by the C1CL. Because of its finite size and associativity, the buffer is prone to thrashing when the buffer size is too small to hold the *Active Critical Data*. **Fig. 6.1** and **Fig. 6.2** show the size of critical data set for some of the benchmarks. For many benchmarks, the size is < 100 words (400 bytes). Therefore most applications whose workload is represented by these benchmarks, are expected to have their active critical data set less than

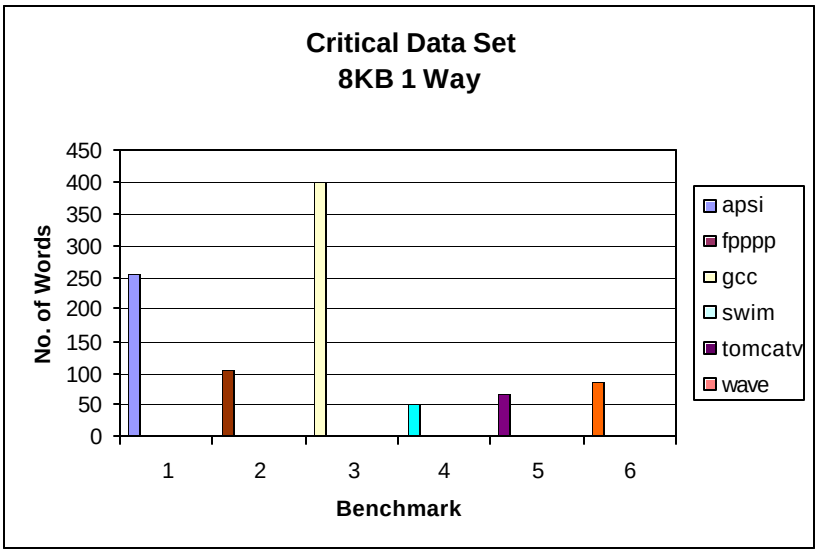

Fig. 6.1

2KBytes. Since this is quite a small buffer, it can be made highly associative without affecting the critical path and this would decrease the probability of thrashing due to conflicts.

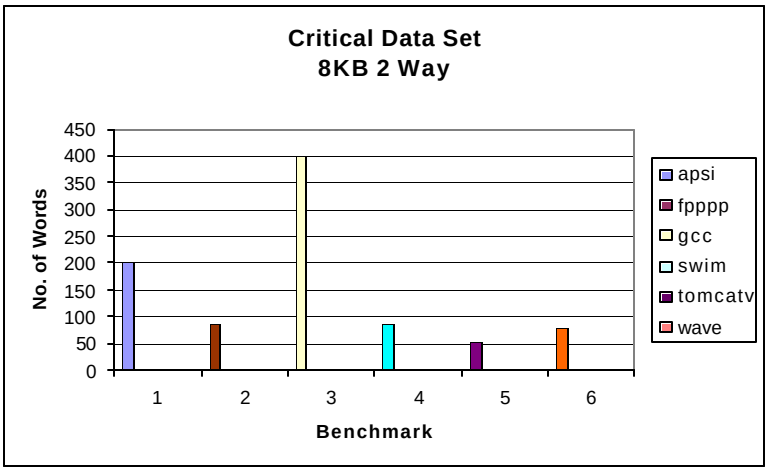

Fig. 6.2

6.2.2 Buffer Pollution

One would like to keep the data accessed by C1CL in the buffer but not the rest of the critical loads i.e. the buffer has to be prevented from being polluted by other loads. It is very hard to determine dynamically whether a load belongs to category 1 or not. The category 1 loads can be identified statically (using traces) and the loads in the executable can be tagged with this extra information. This information can be used to determine whether to allow data accessed by this load to reside in the buffer or not. This technique will effectively tackle the problem of buffer pollution.

7. SUMMARY

In this report, we presented an analysis of the access patterns of the loads that are critical to the performance of an out of order processor. Our measurements reveal that 30%-40% (an in some cases, upto 98%) of the critical load instructions access the same memory location they accessed the last time they were executed. Moreover, in more than 80% of the cases, the successive occurrences of a critical load instruction in the dynamic instruction stream are atleast 512 instructions apart. On the basis of above analysis, we presented a memory management technique to decrease the latency of these critical loads and speedup the overall performance of an out of order processor. This scheme utilizes a Buffer to keep the data accessed by a significant fraction of the critical loads. Our measurements indicate that for most of the Spec95 benchmarks, the size of the Buffer required is much smaller than the L1 cache.

8. FUTURE WORK

A functional simulator was used in this project for identification of the critical loads. The use of a detailed timing simulator like *sim-outorder* can provide a *more accurate* identification of critical loads and more data for further insights into the behavior of critical loads. Further work includes an implementation of the memory management technique proposed in this report and measuring the performance gain.

9. ACKNOWLEDGMENTS

We would like to thank Prof. David Wood for his guidance through this project and Avinash Sodani for the initial motivation of this project and his regular advice and suggestions.

10. REFERENCES

1. S. Srinivasan, Alvin R. Lebeck, *Load Latency Tolerance in Dynamically Scheduled Processors*, 31st Annual ACM/IEEE International Symposium on Microarchitecture, 1998.

2. Santosh G. Abraham, Rabin A. Sugumar, Daniel Windheiser, B.R. Rau and Rajiv Gupta, *Predictability of Load/Store Instruction Latencies, Proceedings of the 26th Annual International Symposium on Microarchitecture*, pp. 139-152, December 1993.

3. Teresa L. Johnson and Wen Mei W. Hwu, *Run-time Adaptive Cache Hierarchy Management via Reference Analysis*, Proceedings of the 24th Annual International Symposium on Computer Architecture, page to appear, June 1997.

4. D.C. Burger and T. M. Austin, *The SimpleScalar Tool Set, Version 2.0*, Computer Architecture News, 25 (3), pp. 13-25, June 1997, Extended version appears as UW Computer Sciences Technical Report #1342, June, 1997.

5. S. Srinivasan, Alvin R. Lebeck, *Exploiting Load Latency Tolerance in Dynamically Scheduled Processors,* Technical Report, CS-1998-03, February 1998, Computer Science Department, Duke University.